RetroSkeleton: Retrofitting Android Apps

Benjamin Davis
University of California, Davis
bendavis@ucdavis.edu

Hao Chen
University of California, Davis
chen@ucdavis.edu

ABSTRACT

An obvious asset of the Android platform is the tremendous number and variety of available apps. There is a less obvious, but potentially even more important, benefit to the fact that nearly all apps are developed using a common platform. We can leverage the relatively uniform nature of Android apps to allow users to tweak applications for improved security, usability, and functionality with relative ease (compared to desktop applications). We design and implement an Android app rewriting framework for customizing behavior of existing applications without requiring source code or app-specific guidance. Following app-agnostic transformation policies, our system rewrites applications to insert, remove, or modify behavior. The rewritten application can run on any unmodified Android device, without requiring rooting or other custom software.

This paper describes RetroSkeleton, our app rewriting framework, including static and dynamic interception of method invocations, and creating policies that integrate with each target app. We show that our system is capable of supporting a variety of useful policies, including providing flexible fine-grained network access control, building *HTTPS-Everywhere* functionality into apps, implementing automatic app localization, informing users of hidden behavior in apps, and updating apps depending on outdated APIs. We evaluate these policies by rewriting and testing more than one thousand real-world apps from Google Play.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

Keywords

Android; Dalvik; VM; bytecode; rewriting

1. INTRODUCTION

Android is now the dominant smartphone platform for new devices sold, with a 68.1% market share of smartphones shipped in the second quarter of 2012 according to the International Data Corporation Worldwide Quarterly Mobile Phone Tracker [20]. A major feature of the Android platform is the tremendous number of third-party Android apps available for users to install on their devices, with more than 700,000 Android apps [26] available on Google Play [5] alone.

Many Android apps have access to a wide variety of sensitive information, including banking and payment details, email and messaging content, photos, and browsing history. It may be difficult for a user to determine exactly what an app may do, so the user must rely on their limited assessment of the trustworthiness of the developers of the app, and of any included third-party libraries. Most users with devices associated with a telecommunication provider either can not or do not root their devices, which can be a risky and unsupported operation. Without access to privileged and low-level functionality, these users are left with even less control over what apps may do on their device than on traditional desktop systems. For these reasons, commonplace practices on traditional desktop PCs, such as installing a third-party firewall to monitor incoming and outgoing connections, cannot be performed on most stock devices. The controls provided by the Android platform, such as the permission system, are extremely limited, coarse-grained and inflexible. For example, if a user installs an apps that requests the INTERNET permission, then the app is granted permanent, unlimited access to the network.

Our work is based on the observation that the vast majority of Android apps (estimated over 95% [28]) are entirely compiled into Dalvik bytecode (i.e., containing no native code). Not only are most apps built on the same framework, but the Dalvik bytecode does not include the ambiguity that frequently makes analysis of other compiled software (e.g. x86 machine code) impractical. We take advantage of the consistency in app implementation to design and implement an Android app rewriting framework for customizing behavior of existing applications without requiring source code or app-specific guidance. We call our app rewriting system RetroSkeleton after the ability to retrofit apps with new behavior by modifying their internals.

RetroSkeleton can be used to provide users with more control over the security, usability, and functionality of apps, even compared to desktop applications. We provide a system for automatically embedding policies completely into applications via automated app rewriting to insert, remove, or modify app behavior. Our system is capable of supporting a wide range of flexible, customizable transformation policies that can be written and applied to apps automatically without any app-specific knowledge or guidance. In this paper we describe several different policies we have implemented and applied to real apps, providing features such as fine-grained network access control, HTTPS-Everywhere features, or on-the-fly localization of app UI elements. Apps transformed

MobiSys'13, June 25–28, 2013, Taipei, Taiwan

by our rewriting system can be installed on stock, unmodified and unrooted Android devices.

Our Android app rewriting system was designed to have four important properties:

- **Complete**: intercept all target method invocations, including dynamic invocation via reflection
- **Flexible**: powerful enough to support a variety of complex transformation policies
- **App-Independent**: no manual app-specific guidance needed to create or apply transformation policies
- **Deployable**: rewritten apps work on unrooted, unmodified Android devices, no additional software required

1.1 Threat Model

We assume that the user of a rewritten app trusts the Android platform, the transformation policy writer and rewriting process, but not the original app code. We do not attempt to hide the fact that the app has been modified from the rewritten app.

1.2 Contributions

In this paper, we make the following contributions.

- Design and implementation of an Android app rewriting system capable of replacing arbitrary method calls with custom handlers
- Flexible framework supporting the application of app-agnostic transformation policies to arbitrary Android apps
- Sample transformation policies, including the integration of fine-grained access control, improved security of network communications, and automatic localization of UI elements
- Application and evaluation of these polices to over one thousand real-world apps available via Google Play

A major contribution of our system is the ability for policy-writers to specify high-level policies that can be applied to arbitrary apps automatically without any app-specific guidance. Policy writers identify the Java methods of interest and write Java source code, or higher level functions that generate Java source, for the bytecode operations they wish to embed into rewritten apps. Our system automatically generates all handler classes, methods, and supporting code to intercept all method calls of interest, even those invoked via reflection. Our system can apply a single policy to any real-world app automatically without requiring app-specific guidance.

2. DESIGN

We change application behavior by intercepting method invocations in Android apps. In our system, we call the methods we intercept the *target methods*. Wherever the original app would have invoked a target method, our rewritten app instead calls a new method we've added to the app. We call these new methods we add to rewritten apps method *handlers*. Generally, each target method has its own corresponding handler. The *transformation policy* includes the list of target methods and the Java source code for each associated handler method. A person we call the *policy writer* creates a transformation policy that can be applied to any arbitrary app (these policies are app-agnostic). A *user of our rewriting system* provides a transformation policy and an Android app as input to our rewriting system. Our rewriting system analyzes the app, augments and applies the transformation policy without any user guidance, and produces a new, stand-alone Android app suitable for deployment to an unmodified Android device. We include a high level view of the components of our rewriting system design in Figure 1.

2.1 Approach: Method Interception

Typical policies regulate how apps interact with the sensors, data and other resources of the system. We intercept method calls because they are the primary mechanism that Android apps use to interact with the underlying device. For example, apps use method calls into the Android platform API to send and receive data over the network, access the GPS and other sensor data, read and write data to the address book and other stores, and manage the UI. Policy writers can take advantage of existing knowledge of the Android platform when designing transformation policies, rather than building policies around the implementation details of how certain app behavior may manifest itself in low-level system calls invoked by the Dalvik virtual machine.

2.2 Identifying Method Invocations

Most Android apps are written in Java, and compiled down to bytecode that runs on Android's Dalvik virtual machine. While the architecture is different, Dalvik bytecode is at a similar level of abstraction to that of the Java virtual machine. Specifically, instructions in Dalvik bytecode are clearly defined by the specification and can be interpreted without the ambiguities that complicate the analysis some other platforms such as x86 machine-code.

In Dalvik bytecode there are several instructions for method invocation. Method call instructions include the invocation type (e.g. static, direct, virtual, super), the method signature (which includes the parent class, method name, and parameter types), and the registers associated with the call. It is not always possible to identify the exact method that will be executed at runtime, due to virtual method invocation, inheritance and related concepts. For example, a method call instruction may specify a method signature for the parent class of the object on which the instruction is invoked at runtime. If the object is the instance of a class that extends the parent class and overrides the method in question, a virtual method invocation specifying `Parent->method` will execute `Child->method` at runtime. There are many cases that much be addressed when building a general system for identifying and intercepting method calls of any type, which are complicated with issues such as protected methods, final methods, and so on. So while it is easy to identify method-invocation instructions, we must adopt a number of strategies to ensure that we intercept all target method invocations despite these challenges.

2.3 Method Handler Design

Our strategy for method interception is based on our previous work for a system called I-ARM-Droid [11]. This previous work describes method interception as a means for implementing inline reference monitors in Android apps. We leverage the method interception strategy from this earlier work to build our generalized and complete rewriting system. In this section we give an overview of the strategies involved with intercepting different types of method invocations.

We intercept method invocations by identifying all of the instructions that may invoke a target method, and replace them with instructions to invoke one of our handler method and store the result in the appropriate register. These method handlers may perform any operations described in the transformation policy, which may include invoking the original method call and returning the result (e.g. after notifying or asking the user). To support this kind of dy-

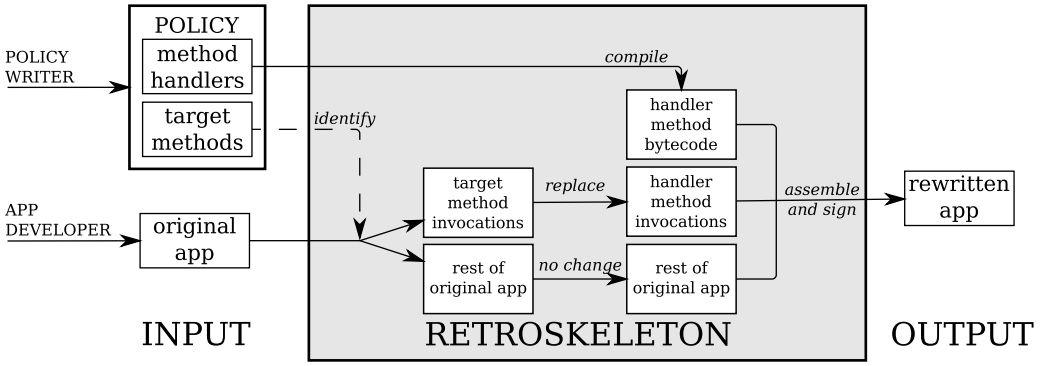

Figure 1: RetroSkeleton System Diagram

namic runtime introspection in our method handlers, our rewritten applications pass all of the original instance objects and parameters involved in the target method call to our handler methods. We require a number of different approaches to handle all of the ways target methods may be invoked.

2.3.1 Stub Methods

Static Methods We replace calls to a public, static target method (such as the `loadLibrary` method in `java.lang.System`) with a call to a static, public handler method that takes the same parameters. We call these static, public handlers *stub* methods. Our stub methods receive all of the same parameters as the original target method would have, and returns a value of the same return type. Because the stub methods receive all of the original parameters, the handler may invoke the original target method from within the handler if desired.

Instance Methods Normal instance method invocations can also be replaced with calls to stub methods. For example, imagine an app invoking the `openStream` method on a `java.net.URL` object instance. Because the `openStream` method is final, we know that the object must be a `URL` object (as opposed to some child class that has overridden the method). We replace this method invocation with an invocation of a static stub method. However, we also pass the instance on which the target method is invoked to the stub method handlers for instance target methods. In our example, our static stub handler for `openStream` receives the `URL` object as a parameter, which it may inspect and perhaps invoke the original target method from within the handler, as specified by the transformation policy.

Constructors We replace invocations of target constructors with calls to our associated static stub "factory" methods that receive the same parameters as the original target constructor. Inside these static handlers, we create an object of the appropriate type and return the object back to the original application. We also insert a move instruction that copies the reference to the new object returned from the static handler into the register on which the constructor was originally invoked.

Occasionally, this approach can result in bytecode that does not pass verification. Dalvik bytecode is register-based, with "virtual" registers local to each method. Constructors are invoked on registers containing uninitialized references to an object, and the object is initialized in place. However, with our approach, bytecode restrictions prevent us from passing uninitialized references to our factory methods, so we create a new object and return the new reference. If two registers contain the same reference to an uninitialized object, and we replace the constructor invoked on one with a call to our stub method, we update one register with the new object but the other register still refers to the uninitialized object. The bytcode verifier can detect when later instructions in the method may refer to the uninitialized object and will prevent the app from executing on an Android device.

While this issue is responsible for the majority of app failures in our evaluation (see Section 5.2.2), it is still relatively rare in practice (detected in less than 7% of apps tested with our policies). We could resolve this issue by analyzing the register usage in the same way as the bytecode verifier and copying our new object into all necessary registers, but we have not yet implemented this functionality.

2.3.2 Wedge Classes

While the simple stub-method-based approaches described above cover many cases, we need a separate technique to handle the use of inheritance and virtual method invocations frequently found in Android apps. We handle the remaining cases by analyzing the classes declared in each Android app during rewriting, and injecting classes providing handler methods into the object hierarchy of the original app.

Developers of Android apps often create new classes that extend existing classes. Assume we wish to intercept the `bind` method for the `DatagramSocket` class in the `java.net` package. Consider the case when an app developer creates a `DevSocket` class that extends `DatagramSocket`. The way we handle the invocation of the `bind` method on an instance of the developer's class depends on the goals of our policy.

Full-hierarchy interception Our system supports full-hierarchy interception, which means that for any class A with target method m in our transformation policy, our rewritten app should also intercept all invocations of m on objects of type B where B extends A. Our rewriting system analyzes all classes in an app and identifies all classes that extend target classes specified in the transformation policy. For these cases, our system can automatically add B's m to the list of target methods and generate the associated handler code and rewrite as normal.

Partial-hierarchy interception In practice, however, we find that it is often more useful to only intercept method calls that invoke the method in the parent class. For example, many Android platform methods provide the API to low-level interactions with the underlying device, such as sensor, network, and file system access. Many useful transformation policies control the app's ability to interact with the API in the framework, but are not concerned with intercepting developer methods that override target methods without invoking the original target method.

For example, if the developer creates a `DevSocket` class that extends `DatagramSocket` and overrides the `bind` method with an implementation that performs no network operations, a transformation policy controlling network access would not want to intercept this method. On the other hand, if the developer overrides `toString` with code that invokes `super.bind` then we would want to intercept this request.

We provide this functionality by generating *wedge classes* that extend non-final classes containing target methods, and include *wedge method* handlers for each target method in the parent class. In our example, our wedge class would extend `DatagramSocket` and override `bind`, implementing the handler method behavior. Our rewriting system also identifies all classes extending the target class, and modifies them to instead extend our wedge class, and replace all internal calls (e.g. via `super`) as appropriate.

In our rewritten app, invoking the `bind` method on an instance of `DevSocket` only invokes our handler when `DevSocket` has not overridden `bind`, or if the developer's class calls `super.bind` from within the class. This allows us to properly intercept all and only calls to the platform target methods, without intercepting developer methods that do not invoke our target methods. This approach is also useful for target methods that cannot be directly invoked, such as methods marked with the `protected` access modifier.

2.3.3 Use of Stub Methods and Wedge Classes

We use both the stub and wedge class approach together so all target methods are intercepted however they are invoked in the rewritten app. Static, direct, and virtual method invocations on target methods are rewritten to stub method calls. Virtual and super invocations on non-target methods that resolve to target methods are intercepted by wedge methods. We do not generate stub or wedge methods in cases where they are unnecessary. Specifically, we do not wedge final or abstract methods or create stub methods for target methods that cannot be called publicly. Section 3 will describe how we handle special cases, such as reflection.

2.4 Method Handler Behavior

As described above, transformation policies primarily contain two types of information. First, they contain a list of all target methods, identified by method name, containing class, parameter types, along with their associated data such as the return type, attributes (protected, final, etc.) and checked exceptions declared. Second, transfer policies describe the method handlers, which is what executes whenever the original app would have invoked a target method.

In order to preserve application functionality as much as possible, our handler methods return values matching the return type of the associated target method. When the target methods are replaced, the parameters (and instance object, when applicable) normally passed to the target method will be passed as parameters to the handler. Because the parameter and return types of the target methods are specified in the transformation policy, along with the appropriate attributes of the method (e.g. static, protected, final), our system generates the appropriate signatures for all stub and wedge methods. The policy writer only needs to specify the body of these handler methods.

The method handler bodies are written in Java source code. This design allows policy writers to work at the level of abstraction Android apps are normally implemented, and apply existing platform knowledge, experience and tools to guide policy development. In order to preserve app behavior when possible, we require and confirm as part of the handler method compilation process that handler methods return a value of the type the target method originally returned. We also recommend that handler methods do not throw new exceptions that the original app would not expect from an invocation of the target method, though we do not enforce this.

While policy writers may specify the source code for handler methods directly as a string, we also allow policy writers to specify functions that generate the appropriate Java source to alleviate the tedium of writing similar handlers for many methods (e.g. logging a set of target methods to the Android logging system). In many policies, method handlers may invoke the original target method, perhaps after logging the request or performing an access control check. We call the expression for invoking the original target method from within the handler method (using the handler method parameters) the *passthrough* behavior. Similarly, method handlers may wish to avoid calling the original target method, but must return a value to the application in order to preserve app behavior. Our system can also generate a reasonable default "deny" expression based on the attributes of the target method. For target methods that return void, the handler may simply `return`. A handler method could create and throw a new exception matching a type declared by the target method. If there are no better alternatives, the handler could simply return `null`. Our system allows policy writers to specify handler-generator functions that receive a map of the target method properties, auto-generated passthrough source, and auto-generated deny source at rewrite-time, and return Java source. Policy writers are free to use or ignore the default passthrough and deny behavior.

We also provide a separate namespace for policy writers to create support classes for code outside of the handler method classes. This allows policy writers to centralize and avoid duplication of code across multiple handler methods.

2.5 Generating Support Code

In Android apps, all of the Dalvik bytecode is combined into a single `classes.dex` file. All of our handler methods are specified in Java source, but without the source code for the app we cannot use the standard Android app development tools to recompile the entire app. Instead, we generate the Dalvik bytecode for our classes containing handler methods, then merge the bytecode with the rewritten application bytecode before assembling the rewritten app.

We use the Android development tools to create a new, nearly empty Android project. We generate the source code for the classes containing our handler methods into this project directory. All of our handler classes are placed into a unique namespace that will not collide with any classes in the original app. In each of our wedge classes, we also generate constructors corresponding to each constructor in the parent (target) class. Constructors are not automatically inherited, so if a wedged class is expecting a particular constructor to be present in the parent class, we must provide it in our wedge class. These constructors simply invoke the associated target classes' constructor unless otherwise specified in the transformation policy.

If the transformation policy has been augmented with target methods found in the original app, then the handler methods may have developer classes used in the parameters or return type of these target methods. If these developer classes do not exist in our support project, then the handler methods will fail to compile as the types referenced will not be found. We do not have source code for the original app's classes, so instead we simply generate *skeleton classes* with the necessary types and methods. These skeleton classes will not be included in the rewritten app, and are only used so the Android build tools can compile the source for the handlers

```
(def mytargets
 (merge
  (mktarget
   ["Ljava/net/URL;" "openStream" []]
   {:return "Ljava/io/InputStream;"
    :exceptions ["java.io.IOException"]
    :attributes #{:final}
    :stub-policy policy-logcat})

  (mktarget
   ["Landroid/app/Activity;"
    "onResume" ["Landroid/os/Bundle;"]]
   {:return "V"
    :attributes #{:protected}
    :wedge-policy policy-activity-update})))
```

Figure 2: Example Target Method Specification

```
(defn policy-logcat
  [[contain-cls method-name param-types] _]
  passthrough]

  (str "android.util.Log.i(\"RWAPP\", \""
       contain-cls "->" method-name
       "(" (join param-types) ")"
       "\";\n")
       passthrough))

(defn policy-activity-update
  [target passthrough]

  (str "rs.ActivityMon.setActivity(this);\n"
       (policy-logcat target passthrough)))
```

Figure 3: Example Policy Implementations

and generate valid Dalvik bytecode with the appropriate type signatures.

We use the official Android development tools to compile and build an app containing Dalvik bytecode for all of our handler classes. We disassemble the bytecode of this app and extract the bytecode for all of the stub, wedge and other support classes included in the policy. These classes exist in a package that will not conflict with existing code in the app being rewritten. We merge the bytecode for these classes into the app we are rewriting, and then assemble the result into a new `classes.dex` file including all of the rewritten versions of the original app code and our handler classes. We use the smali [18] Dalvik bytecode assembler/disassembler to build our modified bytecode.

2.6 Transformation Policy Design

Policy writers specify policies in Clojure [3] (a Lisp dialect). The main part of the policy is the description of the target methods and method handler generators. The targets are specified as a map of the tuple of the containing class, method name, and parameter types of the target method, to a hashmap of various attributes including the return type, checked exceptions (if any), non-default attributes (if any) and the stub and wedge policies. We include a very simple definition in Figure 2. This policy logs every time the `URL->openStream` method is invoked, and logs and updates a global reference to the current Activity whenever `Activity->onResume` is invoked. Note that `URL` class is final so its target methods are not wedged, and the `onResume` method in `Activity` is protected, so it does not have a static stub method.

The `mktarget` function generates the stub and wedge class names and other default values. While policy writers can create custom `mktarget` functions to set and automatically apply default stub and wedge policies, we specify them explicitly in our example. Stub and wedge policies are specified as functions that receive the tuple describing the target method and the passthrough source and produce the Java source code for the method handler. We also provide a helper function that passes the automatic deny code as well, though we do not show this here.

The `policy-logcat` function produces the Java source to log the containing class, method name, and parameter types to the Android logcat logging facility with the tag "RWAPP" before invoking the passthrough code. The `policy-activity-update` function produces the Java source to update global reference to the current Activity by passing the value of `this` to some of our support code, before logging the request and invoking the passthrough code. See Figure 3 for the code listing for these two policies.

3. CHALLENGES

3.1 Reflection

Android apps can invoke methods specified dynamically via Java's reflection API. While the precise methods invoked via reflection cannot always be identified statically, we *can* identify the calls to the reflection API statically. For example, we statically identify and rewrite calls to Java's `Method->invoke` reflection method even though the parameters may be determined at runtime.

Our handler methods for the reflection API perform dynamic inspection of the parameters to determine at runtime the method the app is attempting to invoke via reflection. If it matches one of our target methods, we instead invoke the associated target handler and return the result, otherwise our handler performs the requested invocation as expected. Our reflection handlers are recursive in the sense that they can handle attempts to invoke the reflection API via reflection.

3.2 Native Code

While it is relatively uncommon (estimated less than 5% by Zhou et al. [28]), Android applications may also include native code. Native code is compiled into machine-code specific to a particular architecture, and the techniques for analyzing and instrumenting this kind of code is outside the scope of this work. While we do not attempt to make any guarantees about what native code may do once invoked, we detect and prevent the invocation of native code in rewritten apps by identifying the invocations of native code. Before rewriting an app, we analyze the bytecode and extract signatures for all of the native methods. We augment the transformation policy by adding these native methods to our targets list and generate method handlers for each. For example, in our fine-grained network access control policy we automatically generate handlers that log these calls and ask the user whether to allow or deny the call to native code at runtime. We do not attempt to rewrite Android apps written entirely in native code.

3.3 Intercepting Unexamined Code

Android apps have several mechanisms for dynamically generating or loading classes, bytecode, or libraries. We include signatures for the methods that provide these capabilities to prevent rewritten

apps from executing unexamined code. Policy writers can specify handlers for these methods that can block or allow these requests dynamically based on runtime decisions (e.g. input from the user).

3.4 Integration with App UI

By embedding our policies directly into our rewritten apps, we have the ability to achieve deep integration with the operation of the rewritten app. It is particularly useful to be able to integrate into the UI of the rewritten app. In Android, most operations involving the UI must be made within the main (UI) thread, with a reference to the current Context (such as the foreground Activity, Android's name for a GUI frame). These features may not be easily accessible within all method handlers (e.g. invoked from a background thread).

In order to simplify UI-based operations made from within method handlers invoked from a variety of contexts, we have implemented a small support class and set of method handlers for various Activity lifecycle methods (e.g. the `onCreate` and `onResume` methods) that maintain a globally accessible model of the current Activity displayed to the user. This context can be used to execute events on the UI thread, which we use to display notifications to the user via dialog boxes, popup messages, and integration with app UI elements in our example policies described below.

We build on this to provide a general mechanism that blocks handler methods invoked from a background thread and performing some task in the UI thread before returning. Normally, dialog boxes are requested asynchronously in Android, but we used this mechanism to allow the user to make a decision about how to handle a request in our fine-grained network access control policy. As a convenience, these UI support methods are available to all policy writers, who are also free to implement their own.

4. APPLICATIONS OF REWRITING

To demonstrate the power, flexibility and practicality of our rewriting system, we have developed several example transformation policies. In this section we describe the design and implementation of these policies. We evaluate the application of these policies to real-world apps in Section 5.

4.1 Fine-Grained Network Access Control

In Android if a user installs an app that requests the INTERNET permission then the app is granted unlimited access to make network requests. To provide users with the ability to specify more fine-grained control over the network connections an app can make, we created a transformation policy that intercepts, logs, and prevents apps from making unapproved network operations.

Target Methods Our transformation policy includes target methods for Android platform methods that perform network operations, including methods in the URL, URLConnection, Socket and related classes. Any time the original application would have made a network request, the rewritten version checks to determine if the request should be automatically allowed, denied, or if the user should be asked to decide. If the user is asked, the rewritten app presents a dialog box to the user asking whether to allow or deny the given request. If the network request is attempted in a background thread, the background thread is blocked until the user responds to the dialog window presented in the UI thread. If any request is denied (either automatically by policy, or by user decision) then our handler throws an appropriate exception (e.g. `java.io.IOException`), or returns `null` if the target method would not normally throw an exception. Rewritten apps also log all network access attempts to Android's built-in logging system `logcat`, along with the allow/deny decisions made for each request.

We also intercept network requests made via reflection (see Section 3.1), and prevent unauthorized network access by prompting the user for permission before executing any native or unexamined code (as described in Sections 3.2 and 3.3).

Support Methods We use the mechanisms described in Section 3.4 to present the user with a dialog box when needed to request permission for network requests and the execution of native and dynamically loaded code. To notify the user of requests that were automatically allowed or denied, we display a message via `Toast` message (simple, brief, noninteractive popup message).

4.2 HTTPS-Everywhere

Many popular web services support both HTTP and HTTPS connections. Unfortunately, many of these services, and apps that use these services, default to the HTTP version of these sites. This can leave users vulnerable to sniffing, session-hijacking attacks like Firesheep [9], SSL-stripping and other attacks.

The HTTPS-Everywhere [12] project provides browser extensions that rewrite HTTP web URLs to HTTPS alternatives for many popular services. Often the URL rewriting involves more than simply changing the protocol to `https`. For example, some sites serve static content from Amazon's S3 storage service. While original domain may only serve content via HTTP, the same content can be retrieved via HTTPS from the Amazon S3 servers. The HTTPS-Everywhere contributors maintain a set of manually-specified rewriting rules for many major websites that map resources offered over HTTP to known HTTPS alternatives.

HTTPS-Everywhere is implemented in a browser context where there are well-defined extension systems capable of rewriting URLs. However, this does not help Android apps that make requests directly to these web services, and to our knowledge no such feature is available for Android apps. Researchers such as Fahl et al. [16] show that many Android apps fail to use HTTPS even when it is available. We can use our rewriting system to produce apps that use the HTTPS version of URLs when available using the URL-rewriting rules from the HTTPS-Everywhere project.

Target Method Handlers We have ported popular URL-rewriting rules from the HTTPS-Everywhere project into support methods we add to rewritten apps. We created a transformation policy that intercepts many platform methods for creating network connections to a URL. In the handlers for these network API methods, we replace (at runtime) the destination with the HTTPS alternative when we have an applicable HTTPS-Everywhere rule.

4.3 User Notification of Background Activity

We can write transformation policies that make use of our deep integration with the rewritten app. We have written a transformation policy designed to inform users of background network requests by augmenting the app UI itself.

Target Methods Handlers In this transformation policy, when we intercept a network request we update the count of requests and display the new total in the title of the current Activity on the screen. We use the network target methods from the fine-grained access control policy and keep track of the current Activity using the mechanism described in Section 3.4. After updating the total, our handler methods create a new Runnable task to execute in the main thread and update the app UI.

4.4 Automatic App Localization

Our rewriting system can also be used to perform some forms of localization and internationalization automatically. We have created a policy that rewrites apps to translate text in app UI widgets via an online web translation service.

Target Methods This transformation policy targets the `setText` and similar methods in some common Android UI classes, including `TextView`, `Button`, `CheckBox`. We intercept `setText` API calls that take strings as well as string asset resource identifiers. We pass the specified string, looking up the string resource if required, to our translation code in our method handlers. One benefit of our approach, as opposed to simply translating all strings found within an Android app, is that our rewritten apps will also translate content generated dynamically, or retrieved from external sources such as the network.

Method Handlers Whenever the original app would normally set the text in one of these widgets, we intercept these calls and translate the text via a web-based translation service, updating the widget with the translated text. Apps usually set the text in UI widgets from within the Android UI (main) thread, so rewritten apps will invoke our handlers in the UI thread. Regular `setText` calls finish quickly, but delaying the UI thread while translating each string via network request would impact the responsiveness of the app. Our system allows us to specify handlers that handle this gracefully.

Asynchronous Design In order to maintain UI responsiveness, our handlers return immediately after creating a new background task to perform the translation. After receiving the translation from the web service, this background task updates the widget with the new text asynchronously, queuing the update task to run back in the UI thread. This means that the app will remain responsive to user and other events while the translation occurs, and the UI will be updated asynchronously as translation results are received.

Our current implementation performs the translation using a simple web service we have created to test our functionality, though the handlers could be updated to function with an existing provider such as Google or Bing. Of course, the translations will only be as good as can be provided by the translation service. We note that when applying a transformation policy that includes network requests, the rewritten app must include the Android INTERNET permission, so this permission must be added if it is not already included in the original manifest.

4.5 Auto-Patching Apps

Our rewriting system can also be used to automatically update apps to run on an updated or different environment. For example, we can write a transformation policy that replaces calls to one API with another. To demonstrate this capability, we created a simple policy that replaces calls to some of the Wallpaper API deprecated in Android version 2.0 to the equivalent functionality provided by the new WallpaperManager. We applied this policy to apps written for the old API and observed that the rewritten apps used the new API. Our system makes it very easy to write these simple policies and have confidence that the target methods will be intercepted, no matter whether the methods are invoked directly or via reflection.

5. EVALUATION

5.1 Evaluation Set Selection

We evaluated the feasibility of our rewriting system for use with real-world apps by testing with randomly selected apps downloaded from Google Play. We have an automated system that crawls and downloads free apps from Google Play, and we selected our apps for evaluation from this set. Some apps available on Google Play fail to install or run, or require specific hardware, so to ensure we would have at least one thousand working apps in our evaluation set we first selected 1200 apps at random. We then discarded apps that were invalid, incompatible or otherwise failed to install or launch on an Android emulator running Android version 4.2. This left us with 1119 apps for our evaluation set. Upon examination we found that of these 1119 apps, only 49 (less than 4.4%) contain native code.

Policy	Total Apps	Failure		Success	
		#	%	#	%
Network	1119	27	2.4%	1092	97.6%
Notify	1119	0	0%	1119	100%
HTTPS	1119	0	0%	1119	100%
Localization	1119	0	0%	1119	100%

Table 1: Successfully rewritten apps

5.2 Rewriting Real-World Apps

We rewrote and tested each of the 1119 apps four different times: once for each of the four general transformation policies described in Section 4. While we confirmed the functionality of our Wallpaper API replacement transformation policy on a few apps known to use the deprecated API, this API was not commonly used in our set of apps so we do not further explore its application here. In this section, we abbreviate the names of the four general transformation policies as follows:

- **Network**: fine-grained network access control
- **HTTPS**: apply HTTPS-Everywhere rules to replace http requests with https equivalents when possible
- **Notify**: display background network activity in app
- **Localization**: translate UI widget content via web service

5.2.1 Rewriting Evaluation

We applied each of our four general sample transformation policies to all of these apps to produce four sets of rewritten apps. Table 1 includes the number of apps that were successfully rewritten when applying each transformation policy. Success at this stage means that the support and handler methods were able to be compiled and integrated into the new app, producing a complete, installable Android Package file (APK).

The Notify, HTTPS, and Localization policies were successfully applied to all apps. The only issues detected in the rewriting stage were when applying the network transformation policy to apps that included native code. Because our network policy is designed to prevent any unauthorized network request, applying this policy automatically adds target methods for each native method in the app. When compiling the handler methods our system generates skeleton classes for the necessary developer classes, as described in Section 2.5. All of the errors in rewriting occur because the target app containing native methods has been obfuscated so that the class and package names in the app are no longer valid Java identifiers (e.g. naming collisions between package and class names). We detect this error and halt the rewriting process, rather than producing a rewritten app that we cannot guarantee has captured all app-specific native methods.

In theory we could detect and resolve these collisions and rewrite our handler code after compilation to match the names used in the rewritten app. We have not yet implemented this feature as it is only necessary when rewriting apps obfuscated in this way with policies that augment the target method list with developer methods. In our evaluation set occurs rarely (2.4% of apps), so we simply detect this case so our system will not produce apps that appear complete but have not been completely rewritten.

Policy	Total Apps	Failure		Success	
		#	%	#	%
Network	1092	6	0.5%	1086	99.5%
Notify	1119	7	0.6%	1112	99.4%
HTTPS	1119	76	6.8%	1043	93.2%
Localization	1119	4	0.4%	1115	99.6%

Table 2: Rewritten apps that successfully verify and run

Policy	Original Size	Mean Increase	
Network	515.3 KiB	38.9 KiB	7.54%
Notify	526.3 KiB	30.8 KiB	5.85%
HTTPS	526.3 KiB	29.6 KiB	5.62%
Localization	526.3 KiB	19.6 KiB	3.72%

Table 3: Mean impact on uncompressed `classes.dex`

5.2.2 App Usage Evaluation

We ran each rewritten app in the Android emulator and manually verified whether rewritten apps pass the bytecode verification stage, launch and begin to execute as expected. We ran each app using the Android monkeyrunner tool and took a screenshot of the running app. In this test, "success" means that the app passed the bytecode verification and ran without crashing, which we confirmed by examining the system and application logs and the screenshot of the running app. While we do not attempt to achieve full code coverage, during testing each successfully rewritten app executed at least one of our added handler methods. General techniques for automatically verifying the behavior of Android apps is an open and difficult problem [23] and may depend on the desired outcome of rewriting. For example, should blocking network access in an app that requires be considered "breaking" the app?

The Network, Notify and Localization policies each had a success rate of over 99%, and the HTTPS policy had a success rate of over 93%. The exact results for each policy can be found in Table 2.

The majority (82%) of the failures occur when Android performs the bytecode verification across the entire app at launch and detects the issue described in Section 2.3.1, when multiple registers in the same method reference the same uninitialized object and a target constructor is invoked on one of these registers. The Android verifier detects this condition in the bytecode before running the application, preventing the rewritten applications with uninitialized registers from executing. It would be possible to perform the same bytecode checks as the verifier at rewrite time and handle this condition by updating both register references, though we have not yet implemented this functionality.

In our evaluation set we observed that the HTTPS policy has the highest failure rate due to its many, frequently-used constructors in the list of target methods. The Network, Notify and Localization policies perform better in our tests as they intercept fewer frequently-used constructors, with mostly non-constructor target methods. For example, these policies intercept the `openStream` method rather than the `java.net.URL` constructor.

The remaining (18%) run failures came from issues involving the application of our transformation policies on apps with unusual characteristics. For example, our localization policy creates a new background thread for each new translation task. While this makes the transformation policy simple to write, apps that attempt to set a large number of widgets at once may cause our rewritten apps to hit the thread pool limit. In our tests our sample policies work well in most cases, and their designs could be made more robust to further improve the success rate.

5.2.3 Transformation Policy Evaluation

In addition to evaluating the rewriting and execution of each app, we also manually verified the functionality of the transformation policies in the rewritten apps.

Network For the network policy, we observed the interception of network requests, dialog boxes presented to the user, and policy decisions applied to network requests.

Notify For the policy notifying the user of background network requests we ran and confirmed the integration and display of this information in the title bar of the current Activity.

HTTPS In our policy we include the URL-rewriting rules from the HTTPS-Everywhere project for several popular sites, including AdMob, Blogger, Google App Engine, Foursquare, Doubleclick, Reddit and related domains. We confirmed the rewriting of http network destinations to https requests via tcpdump when running the apps in the emulator. We transformed http requests to associated https requests in 226 apps (20.2%).

Localization To simplify testing and avoid being tied to any particular provider, we created a simple web service with a to simulate an online translation service. This service receives a string via POST message and returns a translated string. In deployment situations this could pass the string to a web translation service such as those provided by Google or Bing. However, for testing our service returns the string with sentinel prefix and suffix values for easy identification in rewritten apps. We observed our transformed strings in the expected widgets in the rewritten apps, and confirmed the operation of our transformation policy by logging the requests made to our web service.

5.2.4 Impact on App Code Size

We add our handler code as separate methods, rather than injecting our custom behavior inline at every point the original application invoked a target method. This means that even if our handlers are complex and large, they are only included in the app once. Whenever the original app invokes a target method, we replace this single instruction with only one or two instructions to invoke our handler instead (depending on the invocation type). So, the app code increases linearly with the number of handlers in the policy rather than with the number of times an app invokes a target method.

Android apps are packaged in APK files, which contain all of the bytecode and assets for the application. Inside this compressed archive, the `classes.dex` file contains all of the Dalvik bytecode for the application. We measure the impact of the policies described in this paper on the size of an app by measuring the size increase of the uncompressed `classes.dex` file. We note that the `classes.dex` file is compressed in the rewritten APK, further minimizing the impact of our added code.

Table 3 lists the mean size of the `classes.dex` file when extracted from the original (compressed) APKs, and the sizes of the new `classes.dex` files after rewriting but before recompression. Our system successfully applied the Network policy to only 97.6% of the apps, so we compare the sizes of the rewritten `classes.dex` files only to those in apps that were successfully rewritten. The bytecode added scales linearly with the number of target methods specified in the policy and the sizes of the method handlers. This means that a policy that has little variance in the number of target methods specified or handler sizes will add approximately the same amount of bytecode to each app.

Table 4 lists the size analysis on the complete APKs produced by our rewriting system. These APKs are the actual files that are

Policy	Original Size	Mean Increase	
Network	1915.4 KiB	11.03 KiB	0.58%
Notify	1987.1 KiB	9.19 KiB	0.46%
HTTPS	1987.1 KiB	9.66 KiB	0.49%
Localization	1987.1 KiB	7.09 KiB	0.36%

Table 4: Overall impact on APK size (compared to mean size of recompressed original APKs)

deployed to Android devices. We discovered that because several APKs in our evaluation set had been poorly compressed, our system actually produced some rewritten APKs that were smaller than their originals. To discount the effect of different compressions, Table 4 compares the sizes of APKs before and after rewriting both of which were compressed in the same way by us. As in Table 3, we compare our rewritten APKs only to those in the original set that were successfully rewritten.

5.3 App Performance

The impact of our modifications of the performance of a rewritten app depends greatly on the behavior of the handler methods. However, the only overhead required by our design is the addition of one new method call for each target method invocation in the original app. In [11] we describe our method interception strategy in more detail and perform a microbenchmark test to measure the overhead of adding an additional method invocation for each target method call. We applied a very simple transformation policy that intercepts the `StringBuilder->append` method, with a handler that only invokes the original method, to an app we created that performs one million appends. We ran the original and rewritten versions of the app on a HTC Thunderbolt phone running Android 2.3.4. We measured the run time of each and in this test executing our method handler for each target method invocation adds an average of less than 0.2 microseconds per call. Due to the minor impact on resource usage, this overhead is also unlikely to make a measurable impact on power consumption on the device. Rewritten apps only pay this performance penalty when executing a target method, as the remainder of the app code is left unmodified.

While the inherent overhead of our method interception is low, the overall impact on app performance depends on the functionality of the handler methods and the target methods intercepted. The overhead added by many of our handlers is marginal relative to the original target method call (e.g. logging a network request method). When more time-consuming handlers are required, our deep integration into the app allows for sophisticated policies that minimize the impact on app performance (e.g. our localization policy translates UI text in background threads and updates the UI asynchronously). In general, well-written method handlers will not impact the app performance more than if the original app had implemented the desired behavior. Policy writers can weigh the performance impact against the value of the added functionality.

5.4 Rewriting Speed

Apps can be rewritten quickly on a normal desktop PC. On average, a normal desktop machine with a 2.66 GHz Intel Core 2 Quad CPU (Q9450) with 4 GB of RAM and a single 7200 RPM hard disk our system can rewrite an apk in less than 5 seconds. This is the time required for the entire end-to-end rewriting process, which includes disassembling the original app, analyzing the contents of the target app, generating the Java source for all target method handlers, compiling the handler code, rewriting the app bytecode and merging in the handler code, reassembling and signing the rewritten app. The bytecode rewriting is performed in a single pass, which scales linearly with the size of the app bytecode.

6. DISCUSSION

6.1 Transformation Policy Development

While users may develop transformation policies themselves, it seems more likely that most users will use community developed, vetted, and maintained policies, much in the way that communities have evolved to produce and maintain policies for Adblock Plus [1], NoScript [6], HTTPS-Everywhere [12], and similar security-focused browser extensions. Policy writers can use static and dynamic techniques to analyze the Android platform (such as those described in by Felt et al. [17]) to ensure proper coverage of functionality across the Android platform.

Our system only requires handler methods to declare the same checked exceptions and return type as the target methods, leaving policy writers with the power to modify the state and operation of the app in dramatic ways. This allows for powerful transformations to app behavior, but policy writers must carefully consider the effects of their modifications. It is possible for transformation policies to change the internal state of an app in ways the app does not expect or handle gracefully (e.g. by throwing an unexpected runtime exception). We leave it to the policy writers and users of the rewriting system to decide what changes to app behavior are appropriate and necessary to achieve their rewriting goals.

6.2 Transformation Policy Application

Our system could be deployed in a number of ways. Users may download Android apps and run the rewriting tools themselves to produce a new app to install on their device. Alternatively, this rewriting system may be provided as part of an online service that rewrites Android apps on the fly as requested by a user. Corporate environments may rewrite apps before allowing installation on a device (e.g. via a proxy as apps are downloaded to the device, or by providing a private "market" of rewritten apps).

Imagine an enterprise that requires all apps to be rewritten before installed on a corporate device. This rewriting process could allow the enterprise to mitigate the risk of installing apps developed by untrusted third parties. This enterprise could provide a trusted rewriting service that applies a trusted transformation policy to any arbitrary app requested by a user. This trusted policy can be applied to many apps, so it seems more feasible for an enterprise to maintain a single trusted transformation policy than to develop all of the apps that users desire.

Android apps include a digital signature from the developer over the contents of the app. Of course, because rewriting the app changes the content, the original signature will not be valid for the rewritten version of the app. The rewriting system can generate a new signature for the rewritten app. In this way, users can verify that the rewritten app was created by a trusted rewriting service and was delivered by without tampering.

Rewriting requires just a few seconds on a normal desktop PC, which is fast enough to support on-line rewriting before installation on users devices. While our current rewriting system runs on traditional PCs, it may be possible to perform the rewriting entirely on the Android device itself.

6.3 Advanced Policies

Except when replacing calls to parent classes in wedged classes, our current policies replace each target method with the same handler globally. However, it is possible for us to use our system to differentiate method calls by call site. For example, given the ability to

identify ad libraries in apps, either by package name or class properties, we could rewrite apps to block all network requests made from an ad library and allow all network requests made from the rest of the app. This functionality cannot be provided by solutions lacking the deep app integration provided by our rewriting system (e.g. an ad-blocking proxy cannot distinguish between requests made from ad vs. app code). The blocking of ad requests could be performed as in our fine-grained network access control policy, as well-written ad libraries should gracefully handle what it sees as a failed network request. We leave the exploration of policies that distinguish based on call site to future work.

6.4 Legal and Ethical Discussion

We designed RetroSkeleton to be a powerful tool to enable flexible rewriting of apps. Like many other tools, RetroSkeleton may be used for a variety of purposes. While many uses benefit app markets, developers, and users, some may not. The legality of reverse engineering, modifying, and redistributing apps depends on the jurisdiction and contract (e.g., EULA) of the involved parties. Technically savvy users could disassemble, modify, and repackage Android apps without RetroSkeleton, so all the operations performed by RetroSkeleton could be applied manually when RetroSkeleton is unavailable. Furthermore, many effects of RetroSkeleton can also be achieved by other means, such as blocking network requests with a proxy instead of the Fine-Grained Network Access Control policy described in Section 4.1.

Currently, even markets that vet apps before admission make only a binary decision between accepting or rejecting an app. Markets could improve by using automated app rewriting to enforce policies on submitted apps, thus acquiring an additional level of control over the quality of the apps that they distribute to their users. Developers may benefit if app rewriting adds functionality that their users desire without having to implement those features themselves, or in ways that their users find more trustworthy. Users may be more likely to install an app if it has been rewritten using a policy provided by a trusted party to give them additional control or guarantees about its behavior.

On the downside, rewritten apps may restrict behavior that developers or ad providers depend on for revenue (e.g. delivering targeted advertisements). As a result, developers may refuse to distribute their apps on markets that rewrite apps. Users of rewritten apps may not know if undesirable behavior in apps results from rewriting.

While developers may not bypass the interception mechanisms of RetroSkeleton, they are free to detect modifications and change the behavior of their apps (e.g., refusing to run) when rewritten. We make no attempt to hide our modifications from rewritten apps (Section 1.1). Developers can easily check if their apps have been rewritten, e.g. by comparing a checksum of the code with the expected value. Also rewritten apps are signed by the rewriter, rather than the original developer.

7. RELATED WORK

7.1 Modifying the Android Platform

Android includes a permission system that limits apps to categories of functionality declared at install time. Android apps declare the coarse-grained categories of functionality they wish to use in their application manifest. If the user installs the app, the app is granted unlimited access to all data and functionality requested in the manifest. These permissions are very coarse-grained (e.g. a single "INTERNET" permission for network access), and once installed, the user cannot control or constrain what the app does with the granted permissions.

Finding the Android permission system lacking, many researchers modified the open-source Android platform to develop custom builds of Android [19, 29, 22, 8]. Many of these, such as Apex [22] and TISSA [29], are designed to improve the security and privacy controls on Android, modeled around the permission-based methods. While the Android platform is open source, many drivers for hardware support are not, and installing custom Android builds may require rooting the device, voiding the warranty, and/or violating the carrier's terms of service. This makes it infeasible for most normal users to update or modify the firmware or platform to provide additional controls or functionality.

It is also difficult for these custom builds to keep up with new versions to the Android platform, and to support all hardware devices, as open-source versions of device drivers may not be available. In contrast, using our system rewritten apps may be deployed to any stock Android device. In addition to these deployment issues, policies integrated into the platform are far less flexible than our app-specific approach. It is relatively difficult to extend platform-based policies to add new capabilities, and policies in the platform will apply to all apps on the device. Our approach can apply different policies to different apps, and we can easily add and apply new policies at any point.

7.2 Rewriting Android Apps

Jeon et al. [21] developed a system to provide finer-grained permissions for Android. Their work is based on a specialized replacement for some privacy-sensitive APIs and use Dalvik bytecode rewriting to modify apps to use their replacement API. Their replacement APIs fulfill the requests using inter-process communication to shuttle the request to an independent service, also installed and running on the device, which contains all permissions. Their approach shares many advantages with ours, in that it requires no modifications to the underlying Android platform. While their goals are focused on providing these fine-grained permission methods, our work is a more general system capable of replacing any methods of interest with any other custom behavior. We provide a new policy language for high-level app-agnostic transformation policies as well as the analysis and rewriting infrastructure to apply them without manual guidance. We believe our system could be used to implement their approach by targeting the same API calls and providing handlers that perform the inter-process communication to their separate service. Of course, our system is flexible enough to also support other kinds of policies, such as those in Section 4, that embed new behavior into the app itself and do not require inter-process communication to handle intercepted calls.

Other researchers have used app rewriting for even more specialized transformations. Reynaud et al. [24] discovered vulnerabilities in Google In-App Billing by rewriting applications that communicated with the Google Market app to bind to their own fake Market app instead. We believe that RetroSkeleton is general and flexible enough for future researchers to use to perform these interesting investigations without having to develop independent, specialized rewriting systems.

7.3 Alternate Android Approaches

Xu et al. [27] have developed Aurasium, which provides reference monitor capabilities by repackaging Android apps to use a custom version of libc. Their system can enforce security policies by interposing on low-level system calls, and performing their security checks from within the replaced libc call. In contrast, our approach allows policy writers to specify what they want to inter-

cept at the method call level, rather than requiring knowledge of how these operations manifest themselves as low-level libc calls. Furthermore, policy writers specify the new behavior they wish to add as Java code that operates on the Java objects passed to their target method, rather than on the low-level data available within libc functions. Aurasium is designed for reference-monitor capabilities, and we can achieve similar results by intercepting calls to sensitive methods, such as those that require the "INTERNET" permission. However, RetroSkeleton can be used for many other ways, such as app UI augmentation or behavior modification like our auto-localization functionality, which would be far more difficult to achieve from within a replaced libc.

Adblock Plus for Android [2] has recently been released and provides a mechanism for blocking many ads from being retrieved by Android apps. Adblock Plus for Android works by installing an Android service on the device that runs a local proxy server in the background at all times. Then the Android device is configured to use the Adblock Plus local proxy server as the proxy for network connections. This local proxy server blocks requests to known ad services. Our rewriting system may be used to similarly provide ad blocking capabilities, with a number of advantages. First, we can apply different ad-blocking policies for each app, while the Adblock Plus proxy cannot distinguish between call sites. We can rewrite and redistribute apps without Adblock Plus's requirements of a system with manual proxy-specification support (Android 3.1 or newer, custom Android build or a rooted device). Rewritten apps also would not require a proxy service running in the background consuming resources as we can integrate the blocking decisions into the new app. We may be able to provide more advanced blocking, such as blocking network requests from ad libraries but permitting requests made from within the main app, as discussed in Section 6.3.

7.4 Java-based Approaches

While Dalvik bytecode differs from Java bytecode, Dalvik bytecode is created from Java. As a result, tools such as ded [13] and dex2jar [4] have been created to convert Dalvik bytecode into Java bytecode. This approach allows for the use of Java-based analysis tools such as WALA [7] on the resulting Java bytecode. This conversion is nontrivial and frequently produces code that can not be assembled back into a functional Android app, as reported in [24]. While this does not matter for pure analysis-based studies, our goal is to produce valid Android apps after rewriting so we operate on the Dalvik bytecode without conversion.

While the Dalvik and Java virtual machines differ in many ways (e.g. register-based instead of stack-based), some techniques used to rewrite Java bytecode apply to our approach to Dalvik rewriting. Java bytecode instrumentation by Chander et al. [10] involves invocation and class substitution to produce modified Java programs. While our rewriting system works on Dalvik rather than Java bytecode, our stub and wedge class approach for low-level method interception is similar to their Java-based system with respect to the class hierarchy of the rewritten software. RetroSkeleton uses this low-level mechanism to perform aspects of the method interception, and provides a new high-level abstraction that allows policy writers to create policies that can be applied to any app. Without a system automating this work, rewriters must identify and create new and different wedge classes for each rewritten app, depending on the classes used and defined by the original app developer. RetroSkeleton analyzes the app and target methods in the policy to automatically generate the appropriate stub and wedge classes with the necessary properties, and modifies the app class hierarchy as needed. This automated analysis and generation is critical for real-world use given the tremendous number and variety of Android apps available.

Rudys and Wallach [25] apply many of these ideas to develop a more complete reference monitor capable of preventing infinite loops and providing transactional rollback to revert changes made in an aborted operation. While our goals are different, their work demonstrates the power and flexibility of bytecode rewriting to provide advanced functionality to existing apps.

Erlingsson and Schneider used inline reference monitors (IRM) for enforcing security properties in Java applications [15]. The Java IRM design detailed in [14] highlights the many benefits of integrating code into the target application itself, such as easier observation of the internal state of the application. Because our approach embeds all changes entirely into the rewritten apps, our system provides these same benefits to policy writers. Because RetroSkeleton can interpose on method calls of interest, it can be used to embed reference monitor functionality into Android apps, but this is only one potential application of our system. Our system allows policy writers to just as easily write policies that add functionality other than security checks, such as adding new UI behavior.

8. CONCLUSION

In this paper, we have presented the design and implementation of a flexible Android app rewriting framework. This framework leverages the relatively uniform way that Android apps are implemented to provide users with the capability to apply powerful and complex policies to arbitrary apps without any app-specific guidance. Our system rewrites apps to insert, remove and modify app behavior, producing rewritten apps that can be deployed to stock Android devices.

We have demonstrated the power of this approach by implementing policies including fine-grained network access control, HTTPS-Everywhere-like capabilities, informing users of hidden app behavior, and automated UI-element localization. We applied these policies to more than one thousand real-world apps from Google Play with a more than 93% success rate for all policies.

9. ACKNOWLEDGMENTS

This paper is based upon work supported by the National Science Foundation under Grant No. 1018964.

10. REFERENCES

[1] Adblock Plus. `http://adblockplus.org`. Accessed: 2012/12/10.

[2] Adblock Plus for Android. `http://adblockplus.org/en/android-about`. Accessed: 2012/12/10.

[3] Clojure. `http://clojure.org`. Accessed: 2012/12/10.

[4] dex2jar: Tools to work with Android .dex and Java .class files. `http://code.google.com/p/dex2jar/`. Accessed: 2012/12/10.

[5] Google Play. `https://play.google.com/store`. Accessed: 2012/12/10.

[6] NoScript Firefox Extension. `http://noscript.net`. Accessed: 2012/12/10.

[7] T.J. Watson Libraries for Analysis (WALA). `http://wala.sourceforge.net`, 2012. Accessed: 2012/12/10.

[8] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. MockDroid: Trading Privacy for Application Functionality on Smartphones. In *HotMobile*, 2011.

[9] E. Butler. Firesheep. `http://codebutler.com/firesheep/`. Accessed: 2012/12/10.

[10] A. Chander, J. Mitchell, and I. Shin. Mobile Code Security by Java Bytecode Instrumentation. In *DARPA Information Survivability Conference & Exposition II, 2001. DISCEX'01. Proceedings*, volume 2, pages 27–40. IEEE, 2001.

[11] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen. I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications. In *IEEE Mobile Security Technologies (MoST)*, May 2012.

[12] EFF. HTTPS-Everywhere. `https://www.eff.org/https-everywhere/`. Accessed: 2012/12/10.

[13] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th USENIX Security Symposium*, August 2011.

[14] U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, 2003.

[15] U. Erlingsson and F. Schneider. IRM Enforcement of Java Stack Inspection. In *Security and Privacy, 2000. S P 2000. Proceedings. 2000 IEEE Symposium on*, pages 246 –255, 2000.

[16] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 50–61. ACM, 2012.

[17] A. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, pages 627–638. ACM, 2011.

[18] B. Gruver. smali: An Assembler/Disassembler for Android's dex Format. `https://code.google.com/p/smali/`. Accessed: 2012/12/10.

[19] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These Aren't the Droids You're Looking For: Retrofitting Android to Protect Data from Imperious Applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 639–652. ACM, 2011.

[20] IDC. International Data Corporation Worldwide Quarterly Mobile Phone Tracker. `http://www.idc.com/getdoc.jsp?containerId=prUS23638712`. Accessed: 2012/12/10.

[21] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. Android and Mr. Hide: Fine-Grained Permissions in Android Applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 3–14. ACM, 2012.

[22] M. Nauman, S. Khan, and X. Zhang. Apex: Extending Android Permission Model and Enforcement with User-Defined Runtime Constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 328–332. ACM, 2010.

[23] V. Rastogi, Y. Chen, and W. Enck. AppsPlayground: Automatic Security Analysis of Smartphone Applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 209–220, New York, NY, USA, 2013. ACM.

[24] D. Reynaud, D. Song, T. Magrino, and R. S. Edward Wu. FreeMarket: Shopping for Free in Android Applications. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, Feb. 2012.

[25] A. Rudys and D. Wallach. Enforcing Java Run-Time Properties Using Bytecode Rewriting. *Software Security Theories and Systems*, pages 271–276, 2003.

[26] B. Womack. Google Says 700,000 Applications Available for Android. `http://buswk.co/PDb2tm`. Accessed: 2012/12/10.

[27] R. Xu, H. Saïdi, and R. Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 27–27. USENIX Association, 2012.

[28] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, Feb. 2012.

[29] Y. Zhou, X. Zhang, X. Jiang, and V. Freeh. Taming Information-Stealing Smartphone Applications (on Android). *Trust and Trustworthy Computing*, pages 93–107, 2011.